Aluna: Fernanda Rocha de Morais Gonçalves
Matrícula: 2018046920

Introdução

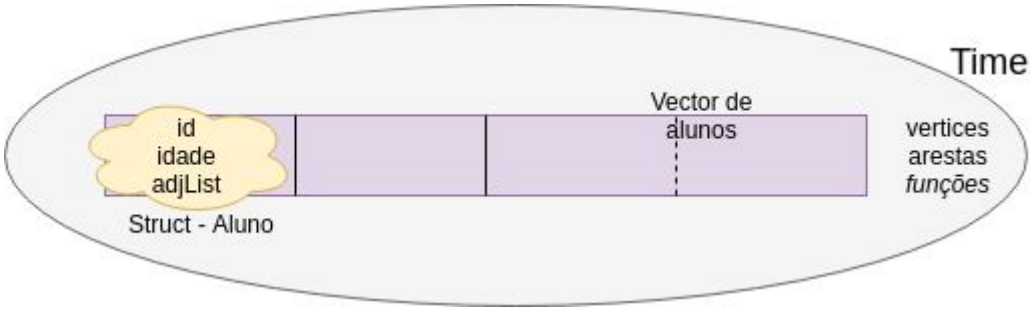
A proposta do trabalho é montar um grafo para representar a organização de alunos da UFMG em subgrupos que contenham representantes, de modo que seja possível identificar quais membros comandam e são comandados por determinados alunos. Além disso, deve ser possível trocar a hierarquia entre os alunos se eles tiverem uma relação direta de comando-comandado, desde que a alteração não resulte num ciclo no grafo.

Ademais, é preciso que dado um aluno seja possível identificar a idade do aluno mais novo que o comanda (direta ou indiretamente) e, por fim, encontrar uma possível ordem de fala entre os alunos que respeite a hierarquia determinada (um comandado não pode falar antes de seu comandante).

Implementação

A linguagem utilizada na resolução do problema foi C++ e o programa foi compilado com o makefile em anexo junto aos arquivos.

O programa possui, além do main, os arquivos time.hpp e o correspondente .cpp. No arquivo .hpp, para a representação dos alunos foi feito um struct com os atributos de identificador, idade e uma lista de inteiros para guardar as relações de adjacência - escolheu-se armazenar para cada aluno a lista de identificadores dos alunos aos quais ele comanda (filhos). Além disso, foi feita uma classe que representa o time de alunos, contendo como atributos o número de arestas (relações) e vértices (alunos) para facilitar posterior acesso e um vector com todos os alunos do time. Os métodos exigidos de SWAP, COMMANDER e MEETING e métodos auxiliares para executá-los também foram declarados nessa classe, uma vez que realizam operações sobre o time de alunos.



Os métodos da classe time estão implementadas no arquivo .cpp, de maneira que a função setVertices recebe como parâmetro o número de vértices totais do grafo e inicializa o atributo correspondente da classe time com esse valor; além disso, recebe o arquivo de entrada na posição para leitura das idades dos participantes e em um laço for, inicializa o identificador e a idade de cada aluno conforme a leitura do arquivo de entrada; em seguida adiciona o participante no vector alunos (pertencente a classe time). O método setArestas também recebe como parâmetro o número de arestas totais do grafo e

inicializa o atributo correspondente da classe time com o valor; o arquivo de entrada também é um parâmetro, mas passado na posição para leitura das relações de hierarquia. Assim, em um laço for o método faz a leitura do identificador do comandante e do comandado e em seguida diminui esse inteiro em uma unidade para ajustar os índices conforme foi inicializado no método setVertices - considerando a posição zero do vector. Por fim, adicionamos o identificador do comandado na lista de adjacência do comandante indicando que há uma aresta direcionada do comandante para o comandado. Nesse ponto, o grafo está montado.

A função executarComandos também funciona na mesma lógica das anteriores: recebe como parâmetro o número de comandos e o arquivo de entrada na posição correspondente. Assim, novamente em um laço for, de acordo com o número de comandos passado por parâmetro, é feita a leitura do caracter que indica o comando e em uma sequência de if-else if chamamos a função correspondente ao comando. Para o comando M, como não há parâmetros, apenas chamamos a função meeting; para o comando C, avançamos no arquivo de entrada para ler o parâmetro do identificador do aluno, é feito o decremento desse número para ajuste já comentado (índice zero do vector) e chamamos a função commander passando como parâmetro o identificador corrigido. Já para o comando S, após a leitura dos dois identificadores dos alunos que deseja-se trocar e o decremento dos mesmos, algumas verificações precisam ser feitas: primeiramente se existe aresta entre os dois alunos; caso negativo já imprimimos na saída “S N” e seguimos com o laço for; caso positivo, chamamos a função de swap e, em seguida, verificamos se formou-se um ciclo; caso negativo, imprimimos “S T” e a troca foi feita com sucesso; caso positivo, desfazemos a troca e imprimimos “S N” indicando que a troca não foi possível.

Sendo assim, passando para as funções de comando apresentadas: para o método meeting foram usadas três estruturas auxiliares sendo uma fila (queue) e dois vectors booleanos. No dois primeiros laços de repetição o objetivo é identificar quais alunos não são comandados por ninguém e, portanto, devem falar primeiro. Para isso, percorremos todas as listas de adjacência de todos os vértices, marcando como true para os índices que encontramos no vector temPai. Em seguida, buscamos aqueles índices que permaneceram o false, ou seja, nenhum aluno o possui na lista de comando. Além disso, esses alunos que não são comandados por ninguém são colocados na fila, de modo que serão os primeiros a falar, pois essa estrutura é caracterizada por ser “First In First Out”. Por fim, temos um último laço de repetição que irá imprimir o identificador do aluno na frente da fila - adicionando uma unidade para a correção da posição zero - e tirá-lo da fila; em seguida, verifica-se a lista de adjacência desse aluno, se o aluno comandado ainda não foi colocado na fila - posição dada pelo índice do aluno no vector visited está como false - marcamos o aluno como visitado e o colocamos na fila, permanecendo no laço até que a fila esteja vazia e todos os alunos tenham sido visitados. Nesse caso, essa implementação funciona pois não podemos ter ciclos no grafo, logo, sempre haverá um ou mais nós que não são comandados por ninguém e assim é possível atingir todos os nós respeitando a condição de hierarquia.

Para a função commander, inicialmente verifica-se se o aluno passado como parâmetro possui algum comandante, procurando nas listas de adjacência dos alunos o

identificador dele e aumentando um contador auxiliar, caso encontrados. Se o contador permanece zero, e então o aluno não possui comandante, imprimimos “C *” e retornamos à leitura de novos comandos. Caso contrário, utilizamos duas funções auxiliares: uma que transpõe o nosso grafo (trocando a orientação de todas as arestas) e outra que utiliza o grafo transposto para encontrar a idade mínima dos comandantes diretos e indiretos do aluno. Chamamos a função idadeMin e a primeira instrução desta é chamar a função que transpõe o grafo para utilizá-lo posteriormente. A função transporGrafo retorna um vector de listas de adjacência com os índices dos alunos, e, para isso primeiramente itera sobre o vector para inicializar todas as posições correspondentes a vértices com uma lista vazia. Posteriormente, para cada aluno, iteramos sobre a lista de adjacência original e trocamos as relações entre eles e, ao invés de termos listas de adjacências com os filhos diretos do vértice, agora teremos listas com os pais diretos dos vértices: para uma aresta (u,v) no grafo original, temos uma aresta (v,u) no grafo transposto. Finalmente, retornamos o grafo transposto.

Sendo assim, de volta à função idadeMin, iteramos sobre a lista de adjacências do aluno de interesse no grafo transposto de forma que os identificadores dos seus pais diretos são colocados em uma fila. Em seguida, inicializamos a variável de idade mínima com a idade de um dos comandantes. Por fim, no último laço de repetição iteramos sobre a fila até que ela esteja vazia, de maneira que pegamos as listas de adjacências dos comandantes inicialmente colocados na fila com o intuito de buscar comandantes indiretos. Para cada comandante indireto, também o adicionamos na fila, para que os seus comandantes sejam analisados. Nesse processo, a cada identificador compara-se a idade com a variável idade mínima, guardando sempre o menor valor, para retorná-la ao fim da função.

Por fim, temos as funções auxiliares para o método swap. A função booleana existeAresta verifica se na lista de adjacência do aluno1 há o identificador do aluno2 e em caso positivo retorna true. Caso negativo, procuramos na lista de adjacência do aluno2 o identificador do aluno1 e em caso positivo também retornamos true. Se não for encontrada correspondência, retorna-se false.

Para o método swap, recebemos como parâmetro os dois alunos e tem-se dois laços de repetição correspondentes: um para verificar a lista de adjacência do aluno1 e outro para a lista de adjacência do aluno2. No laço, iteramos primeiramente na lista do aluno1 procurando pelo identificador do aluno2 que, se encontrado significa que o aluno1 comanda 2. Sendo assim, para trocar a hierarquia, apaga-se o índice de 2 na lista do aluno 1 e adiciona-se na lista do aluno2 o índice de 1 e retornamos para a leitura de comandos. Em caso de não ser achada correspondência no primeiro laço e considerando que já foi verificada a existência de uma aresta, procuramos em seguida o identificador do aluno1 na lista de adjacência de 2, fazendo o mesmo procedimento já explicado mas com os índices trocados.

Já para a verificação de existência de ciclo no grafo temos duas funções booleanas: a temCiclo, que apenas inicializa os dois vectors também booleanos com o valor de false e em um segundo laço de repetição chama a segunda função - buscaCiclo - para todos os vértices, retornando true se buscaCiclo é verdadeira para algum vértice e retornando false se não é encontrado nenhum ciclo; a função buscaCiclo funciona na

lógica da Depth First Search, uma vez que esse tipo de busca em um grafo conectado produz uma árvore e se encontramos uma aresta que volta para um de seus nós antecessores (ou para ele mesmo) então temos um grafo. Para um grafo não conectado, buscamos nas árvores individuais (por isso a busca é realizada a partir de todos os vértices). Para identificar se existe essa aresta que volta para um nó anterior da árvore e forma um ciclo, utilizamos dois vectors auxiliares declarados anteriormente na função temCiclo (stack e visited) e o identificador do vértice que estamos analisando (v); assim, a primeira ação dentro do método é reinicializar a posição de identificador v como true; em seguida, analisamos a lista de adjacência desse vértice e analisamos duas condições: se encontrarmos um vértice que já está no stack temos um ciclo e a função retorna true e se o vértice adjacente ainda não foi visitado mas a chamada recursiva de buscaCiclo já encontrou algum ciclo e retornou true então também retornamos true; Em caso do fim do laço for e nada foi retornado, então não há um ciclo a partir do vértice analisado então tiramos ele do stack e retornamos false.

Tendo explicado até aqui todas as funções da classe Time o entendimento do main é simples. Verifica-se inicialmente se argc é maior do que zero apenas pela necessidade de um arquivo de parâmetro para leitura. Declaramos algumas variáveis e inicializamos a string nomeArquivo com o parâmetro passado; abrimos o arquivo e se não houver erro iniciamos a leitura de acordo com o padrão explicitado pelo enunciado: número de vértices, arestas e comandos. Em seguida chamamos as duas funções que montam o grafo (setVertices e setArestas) e a função que executará os comandos. Por fim, fechamos o arquivo.

Análise de Complexidade

Considerando um número de arestas igual a A e um número de vértices igual a V e que a complexidade para leitura dos comandos não é considerada, as funções serão analisadas separadamente para posteriormente compará-las e estabelecer a complexidade assintótica.

Tempo:

- setVertices = O(V);
  As operações de atribuição dentro do laço de repetição são constantes, assim como a operação de push_back no vector (no pior caso, sendo necessária realocação, a complexidade é do tamanho do vetor).
- setArestas = O(A);
  As operações de atribuição dentro do laço de repetição são constantes, assim como a operação de push_back na lista de adjacência.
- meeting = O(V+A);
  O primeiro laço de repetição é, em uma análise trivial, V(V - 1) pois cada vértice pode ter no máximo V-1 outros vértices na lista de adjacência. Entretanto, a lista de adjacência corresponde a um número menor do que o grau do vértice, uma vez que conta com apenas as arestas que saem do vértice (filhos). Assim, considerando o teorema que diz que a soma dos graus de todos os vértices é duas vezes o número de arestas temos que o laço é O(2A) = O(A). O segundo laço de

repetição possui apenas operações de custo constante, portanto é O(V). O último laço (while) também têm a lógica do primeiro, uma vez que irá analisar as listas de adjacência de todos os vértices e, portanto, também é O(A).

- commander = O(V+A)
  O primeiro laço de repetição também segue a lógica da soma de todos os graus ser igual ao dobro do número de arestas , e, portanto, é O(2A) = O(A).O if que verifica se há comandantes tem custo constante e, como retorna se verdadeiro, no melhor caso a função é O(A).
  - idadeMin = O(V+A);
    O primeiro laço for é O(A) no pior caso, ou seja, se o aluno em questão possui relação com todos os outros. O while irá analisar as listas de adjacência de, no pior caso, todos os vértices, portanto também é O(A). As demais operações de atribuição, comparação, push, front e pop tem custo constante.
    - transporGrafo = O(V+A);
      O primeiro laço de repetição é O(V) pois itera no número de vértices. O segundo laço segue a lógica da soma de todos os graus ser igual ao dobro do número de arestas , e, portanto, é O(2A) = O(A).

- existeAresta = O(A);
  O primeiro laço de repetição é correspondente ao segundo, mudando apenas qual aluno está sendo analisado. Apenas duas listas de adjacências estão sendo analisadas, então em mesmo um pior caso podemos considerar que é um número menor ou igual à soma dos graus de todos os vértices. Portanto, a função é O(A).
- swap = O(A);
  A análise do swap é análoga à função acima, uma vez que os laços de repetição também são correspondentes (com mudança de parâmetro de aluno) e é feita a análise de duas listas de adjacências. Como as operações de erase e push_back em lista tem custo constante, a função têm complexidade O(A) como analisado anteriormente.
- temCiclo= O(V + A);
  Possui apenas um laço para visitar os vértices para achar qualquer ciclo. Portanto, é O(V).
  - buscaCiclo = O(V+A);
    A função funciona na lógica da Depth First Search, sendo assim, visita cada vértice apenas uma vez - nos certificamos disso com o vector visited - e também cada aresta apenas uma vez, devido à característica da lista de adjacência, que guarda a existência da aresta apenas para o vértice da qual ela sai, logo a soma de todos os tamanhos das listas de adjacência de todos os vértices é o número total de arestas - A. Sendo assim, é O(V + A).

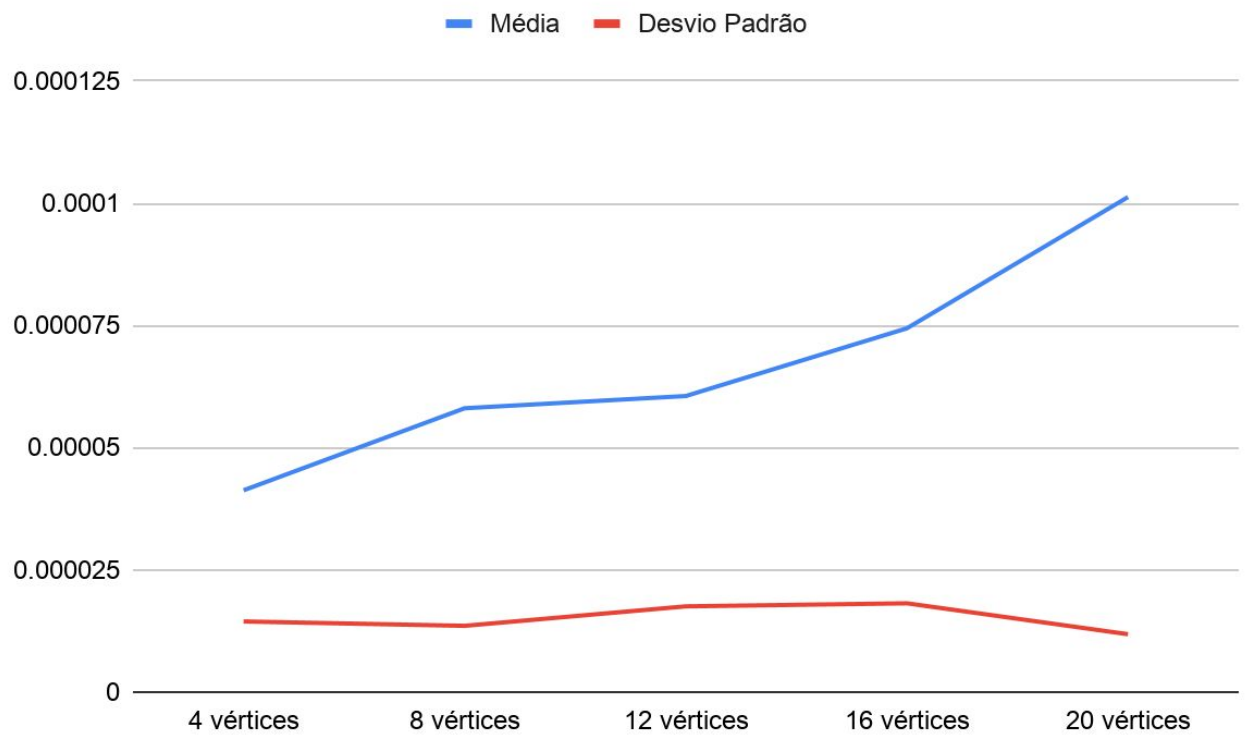
Logo, temos que a complexidade total do algoritmo é O(V+A).

Espaço:

A complexidade de espaço representa a quantidade de memória que é necessário para armazenar as estruturas de dados associadas ao algoritmo. Considerando que para armazenar o grafo precisamos de O(V+A) - temos um vetor de alunos - vértices - O(V) pois para cada vértice guardamos atributos constantes - idade e identificador - e listas de adjacência para guardar as relações entre eles, e, como já analisado anteriormente as somas de todas as listas de adjacência - ou a soma dos graus de todos os vértices - é 2A e portanto O(A). A operação de meeting utiliza dois vetores auxiliares e uma lista, e todos possuem tamanho máximo V. A operação de commander e suas auxiliares utiliza, além da estrutura de vetor de alunos, um vetor de listas auxiliar para armazenar o grafo transposto que precisa de O(V+A) de memória e uma fila com tamanho máximo V. As funções existeAresta e swap não utilizam nenhuma estrutura além da lista de adjacência dos alunos em questão. As funções temCiclo e buscaCiclo utilizam dois vetores booleanos que possuem tamanho máximo V. Logo, temos que a complexidade de espaço final é O(V+A).

Avaliação Experimental

Para a avaliação experimental foram repetidas quinze vezes para entradas com o mesmo tamanho e número de trocas. Foi feito o teste para grafos de 4, 8, 12, 16 e 20 vértices, mantendo constante o número de comandos - 3 commanders, 1 swap possível de ser feito e 1 swap que é desfeito para evitar ciclos e 1 meeting. Segue os resultados da média de tempo e do desvio padrão de execução do algoritmo:



	4 vértices	8 vértices	12 vértices	16 vértices	20 vértices
Média	0.0000412889	0.0000580758	0.0000605752	0.0000744252	0.00010128
Desvio Padrão	0.0000144535	0.0000135615	0.000017544	0.0000181684	0.0000118456

É possível observar que o resultado dos tempos de execução age de acordo com o esperado, aumentando gradativamente conforme o número de vértices.

O grafo deve ser dirigido para guardar as relações de hierarquia de comando e, seguindo a mesma lógica, o grafo não pode ter um ciclo porque a hierarquia seria perdida: um aluno comandaria e seria comandado por todos os outros alunos participantes do ciclo. Além disso, o grafo pode sim ser uma árvore, desde que direcionada, pois essa estrutura de dados respeita às condições dadas na especificação. Entretanto, o grafo não é necessariamente uma árvore, uma vez que, por definição, dois vértices de uma árvore estão conectados por exatamente um caminho, o que não é pré-requisito do algoritmo.

Referências

https://algorithms.tutorialhorizon.com/graph-detect-cycle-in-a-directed-graph/
http://www.cplusplus.com/reference/stack/stack/
http://www.cplusplus.com/reference/queue/queue/
http://www.cplusplus.com/reference/vector/vector/
http://www.cplusplus.com/reference/list/list/